Problem 16: Have You Seen My Key

Points: 35

Author: Marcus Garza, Fort Worth, Texas, United States

Problem Background

In cryptography, there is an encryption scheme which is assumed to be perfect (unbreakable) under some key assumptions, called the one-time pad (OTP) cipher. The premise behind this cipher is that each encryption key is at least as long as the message itself, and once used, is never used again.

Two weaknesses in the OTP scheme are the probability of a codebreaker knowing the type of information being encrypted (the “lexicon”) and the size of the key space. However, even if a codebreaker knows the lexicon, a brute force attack - testing every possible key - is essentially impossible.

Let’s assume you’re trying to encrypt a message that has a key that is 2^{512} bits long. 2^{512} is a BIG number:

13,407,807,929,942,597,099,574,024,998,205,846,127,479,365,820,592,393,377,723,
561,443,721,764,030,073,546,976,801,874,298,166,903,427,690,031,858,186,486,050,
853,753,882,811,946,569,946,433,649,006,084,096

If a codebreaker tried a brute-force attack against this message, and was able to test one key every nanosecond (0.000000001 seconds), it would take 100 trillion trillion trillion trillion trillion trillion trillion trillion trillion years (that’s 9 “trillions”) to test every key. That’s several orders of magnitude greater than the age of the universe, and while it would eventually give you the correct plaintext, it would also give you every other

potentially correct plaintext, making it impossible to determine which message was *actually* correct.

Problem Description

Your program will provide an implementation of the one-time pad cipher. Your program will be given a 128-character hexadecimal string representing the encrypted ciphertext. You will then be given another 128-character hexadecimal string representing the key. The plaintext consists of 64 ASCII characters. To convert the ciphertext to the plaintext, use this process, illustrated in the table below.

1. Get the next two hexadecimal characters from the ciphertext (e.g. ‘4F’)
2. Convert this hexadecimal value to its 8-bit binary equivalent (4F = 01001111)
3. Get the next two hexadecimal characters from the key string (e.g. ‘0C’)
4. Convert this hexadecimal value to its 8-bit binary equivalent (0C = 00001100)
5. XOR (exclusive-or) these two values together to create a new binary number

6. Convert the new binary number to an ASCII decimal value and print that character
7. Repeat with the rest of the message

Hexadecimal 4F = Decimal 79 = Binary 01001111

Hexadecimal 0C = Decimal 12 = Binary 00001100

4F	0	1	0	0	1	1	1	1
0C	0	0	0	0	1	1	0	0
XOR	0	1	0	0	0	0	1	1

Binary 01000011 = Decimal 67 = ASCII ‘C’

If you are not familiar with XOR, it is a logical operation that checks to see if two boolean values are different. If they are, the result is true (1). If they are the same, the result is false (0).

Most programming languages allow you to XOR two numbers together to produce a new number; these “bitwise” operations perform an XOR comparison on each bit of the binary representation of those numbers, just as we did above in the example: 79 XOR 12 = 67. This can be done

XOR	0	1
0	0	1
1	1	0

with the caret (^) operator in Java, C, C++, and Python, and the Xor operator in VB.NET.

Sample Input

The first line of your program's input, **received from the standard input channel**, will contain a positive integer representing the number of test cases. Each test case will include the following lines:

- A line containing a positive integer, **X**, representing the number of keys to use
- A line containing a 128-character hexadecimal string representing the ciphertext
- **X** lines, each containing a 128-character hexadecimal string representing a key

1
2
4F6F0E14089E040E286156061893404F658D1F6510D5744098DB1DF8904D5F0DF23710
D30230F4F985D4FAAE50F4984AF40B4C98F70E98F94998F043DF16D89F
0C006A7128CF716B5B15766F6BB3263A0BAC3F227FBA1060F4AE7E93B0393069934E31
F3515F988FE0F48EC63F87FD6A847923FA9B6BF58A68B8D063FF36F8BF
1B07676728EE686F410F226360E7602704FE3F1178B05433F9B678D8F3242F65974564
B67A44D49BF0A0DACF7090F12C926E3EFD997AB8922CE1DE639E5799DE

(Note that there are only three lines of hexadecimal text above; they're wrapping because they don't fit on the page)

Sample Output

For each test case, your program must output the 64-character plaintexts obtained by decrypting the ciphertext with each provided key, one per line. The plaintexts must be surrounded by brackets. The plaintexts may include trailing whitespace characters, which should be included within the brackets.

[Code Quest is fun! Good luck today! Solve those problems!]
[This plaintext has the same ciphertext but a different key. AAAA]